© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other Works. via https://doi.org/10.23919/DATE48585.2020.9116230

Towards a Qualifiable OpenMP Framework for Embedded Systems

1st Adrian Munera
Barcelona Supercomputing Center
adrian.munera@bsc.es

2nd Sara Royuela
Barcelona Supercomputing Center
sara.royuela@bsc.es

3rd Eduardo Quiñones
Barcelona Supercomputing Center
eduardo.quinones@bsc.es

***Abstract*—OpenMP is a very convenient programming model for critical real-time parallel applications due to its powerful tasking model and its proven time predictability. However, current implementations are not suitable for critical environments based on the intensive use of dynamically allocated memory needed to efficiently manage the parallel execution. This jeopardizes the qualification processes needed to ensure that the integrated software stack is compliant with system requirements.**

This paper proposes a novel OpenMP framework that statically allocates the data structures needed to efficiently manage the parallel execution of OpenMP tasks. Our framework is composed of a compiler that captures the environment of the OpenMP tasks instantiated along the parallel execution and bounds the exposed parallelism, and a runtime implementing a *lazy task creation* policy that significantly reduces the runtime memory requirements, whilst exploiting parallelism efficiently. The evaluation shows that our tool achieves the same performance as current OpenMP implementations, while bounds and drastically reduces the dynamic memory requirements at run-time.

***Index Terms*—OpenMP, qualification, memory allocation**

I. Introduction

Parallel programming models are fundamental to exploit the performance opportunities of the newest parallel embedded architectures targeting critical real-time embedded domains, e.g., the Kalray MPPA [1] featuring 256 cores, the TI Keystone II [2] featuring a 4-core ARM and an 8-core DSP accelerator, and the NVIDIA Jetson AGX [3] featuring an eight-core ARM and a 512 CUDA-cores GPU. In this context, OpenMP is a widely spread model in heterogeneous shared memory architectures from the high-performance domain, that is increasingly being considered in the critical real-time embedded domain [4]–[6]. This is mainly due to its powerful tasking and acceleration models capable of exploiting fine-grain and highly dynamic parallelism, as well as offloading code from the host to an accelerator for boosting performance. Moreover, OpenMP has been included in the software development kits (SDK) of parallel embedded architectures like those mentioned above.

In recent years there has been a significant effort to evaluate the time predictability properties of the OpenMP tasking model. These works are based on the extraction of a Direct Acyclic Graph (DAG) representing the execution of the OpenMP program, a.k.a. *OpenMP-DAG* or Task Dependency Graph (TDG) [7], and upon which timing and schedulability analyses can be applied for both dynamic and static scheduling approaches. However, these works only focus on the analysis of the OpenMP specification [8] without considering the runtime implementations, a fundamental analysis for the adoption of OpenMP in critical real-time environments.

OpenMP has evolved over the last 25 years to cope with unstructured and highly dynamic parallelism in shared-memory and heterogeneous systems. The OpenMP *tasking model* is of particular interest in front of the *thread model*, because it allows to define *what* can be parallel instead of *how* to parallelize it. There, the `task` construct defines an independent parallel unit of work containing a block of executable code (the *task region*) and an associated data environment.

Current OpenMP implementations (e.g., GNU libgomp [9], LLVM kmp [10]) require complex data structures to efficiently orchestrate the parallel execution. Previous works reduce the total memory consumption by statically storing the complete TDG [7]. Still, the runtime makes an intensive use of dynamic memory, complicating the tool qualification and so its potential use in critical real-time embedded systems. The reason is that the *data environment* of tasks has to be captured when the `task` construct is encountered, and it is usually stored in a dynamically allocated structure that can be released when the task completes. As a result, the total amount of dynamic memory allocated at run-time depends on the number of task instances.

This paper proposes a novel OpenMP framework that avoids dynamic allocation of task structures, and reduces and bounds the memory requirements while efficiently exploits the resources, and includes techniques implemented at two levels:
- *Compilation level* : (1) expand the TDG, (2) compute the maximum parallelism, and (3) statically allocate the TDG. The compiler can also initialize the data needed for each task, if this information does not depend on the input data set.
- *Runtime level* : implements two policies regarding the moment at which tasks are created: (1) *lazy task creation* creates tasks when all its dependencies are honored, hence are ready for execution (this technique bounds the amount of on-the-fly tasks, reducing the number of task structures needed to orchestrate the execution to the maximum parallelism expressed in the TDG); and (2) *eager task creation* creates tasks when they are encountered, as most OpenMP runtimes.

We evaluate our framework on three architectures from the HPC and embedded domains (see §IV-A1), executing four real-time applications with high-performance requirements (see §IV-A3). The results show that our framework

(1) provides the same performance speedup than the one provided by the OpenMP framework included in the SDK of the targeted processor architecture, while (2) bounds and significantly reduces the memory requirements, and (3) pushes the use of dynamic memory to the initialization phase.

II. Related Work

Many industries require software to be qualified to specific safety standards (e.g., ISO 26262 for automotive and DO178C for avionics). These define the requirements that systems must fulfill to be safe, i.e., provide proof of absence of critical defects during development and verification processes, including memory issues (e.g., stack overflow, use of dynamic memory), run-time errors (e.g., division by zero, invalid pointer accesses), data races, and violations of timing constraints.

OpenMP tools do not usually take into account safety requirements due to its HPC orientation. However, several works consider safety a major issue even in OpenMP, and tackle the problem from a specification perspective, proposing modifications in order to provide a more robust and safe language, e.g., enhancing *programmability* [11], [12], detecting *data races* and other data environment errors [13], [14], and improving resiliency [15]. More generally, there are works that consider the analysis of the *functional safety* [16] and the *time predictability* [5], [6] of the whole OpenMP specification, both concluding that OpenMP can be functionally safe and time predictable including only minor changes.

However, fewer works consider safety in OpenMP implementations. In this regard, some techniques have been introduced to reduce the amount of memory required by the runtime by statically defining the TDG and avoiding dependence detection at run-time [7]. This work enables the runtime to be used in systems with very limited memory available, such as most embedded systems [17]. Current implementations [9], [10] though still need an amount of memory directly proportional to the number of instantiated tasks (typically high) because tasks are created as they are encountered.

The use of dynamic memory is a major issue when considering qualification because allocation calls are non-deterministic, and their worst-case execution time is either not bounded, or bounded to an excessively large bound, causing an unacceptable predictability of the system [18]. Besides, allocation requests may fail in an unpredictable manner because of several reasons including starvation due to fragmentation or de-allocation, and memory exhaustion among others. For these reasons, certification standards include specific requirements regarding the use of dynamic memory [19], and associations such as MISRA recommend not to use dynamic memory, or to reduce its usage to the initialization of the application [20].

There are other issues regarding the qualification of OpenMP runtimes for safety-critical systems, e.g., the support of wall-clock time, the dependency on POSIX infrastructure, or how system termination in handled. Although this analysis remains as a future work, preliminary analyses on the limitations of OpenMP for qualification have also been conducted under the HP4S (High Performance Parallel Payload Processing for Space) project [21]. This work is in turn aligned with other projects that work towards the adoption of OpenMP in critical embedded systems, such as the support for OpenMP that has been developed on top of the RTEMS Real-Time Operative System [22].

III. Static Allocation of OpenMP Data Structures

This section presents a complete OpenMP framework that allows to (a) statically allocate the memory required by the runtime to execute a program using the OpenMP tasking model, (b) bound the amount of memory needed by the runtime, and (c) bind the memory needed to manage tasks to the maximum parallelism exposed in the application. The techniques, implemented in the two components of our framework (the compiler and the runtime) are detailed below.

A. Compiler Techniques

At compile-time, we have two objectives: (1) define the memory boundaries of the OpenMP runtime data structures, and (2) statically allocate these data structures. We use the Mercurium source-to-source compiler [23] and its built-in infrastructure to generate a complete TDG[1] that represents the order of execution of an OpenMP program based on the `depend` clauses of the `task` constructs, and other synchronization constructs such as `taskwait` or `barrier`.

To determine the size of the runtime data structures needed to efficiently execute an OpenMP program, this paper proposes the next three new compiler features.

1) Capture the data environment: This feature captures (a) the whole data environment of each task instance, if possible (the values of all variables used as firstprivate within the task is known at compile-time), or (b) only the size of the data environment otherwise. If the former is possible, then the data environment can not only be preallocated, but also initialized. In any case, the size of the data environment can always be computed and, remarkably, it is the same for all task instances of a given task construct. This size, together with the size of the task structure, defines the amount of memory needed by the runtime to execute a given task instance.

2) Determine the maximum level of parallelism: This new feature computes the maximum level of parallelism (i.e., the highest number of OpenMP ready tasks that can actually execute in parallel, assuming that enough computing resources are available) based on the shape of the TDG [24]. This work is performed in three steps:

1) Compute the *comparability graph*, which is an undirected graph where it is possible to orient each edge such that the result fulfills the anti-symmetric (i.e., if an edge $u \rightarrow v$, then $v \rightarrow u$ does not) and transitive (i.e., if edges $u \rightarrow v$ and $v \rightarrow w$ exist, then so does $u \rightarrow w$) properties.
2) Compute the *maximal independent sets*. An *independent set* is defined as a set of vertices in a graph, in which none

[1]The compiler can expand a complete TDG as far as the values of the relevant variables are known at compile time, a common feature in applications exploited in critical real-time systems.

of two vertices are adjacent. Based on that, a *maximal independent set* is an independent set that is not a subset of any other independent set.
3) Compute the *maximum degree of parallelism*, which corresponds to the maximum size of the previous sets.

This computation is a NP-hard problem and so an heuristic approach is used [24]. As a consequence, the result of this computation may provide a lower bound of the amount of parallelism actually exposed in the TDG. §III-B presents a runtime mechanism to deal with this case.

3) Static allocation of OpenMP data structures: This new feature considers the analyses performed in the two previous steps to compute the size of the data environment and the amount of parallelism, to allocate the memory data structures that our OpenMP runtime needs to efficiently execute the program. The compiler can allocate the data in two manners:

1) In the data segment (of C programs), meaning that data is defined and initialized statically.
2) In the heap, meaning that the compiler introduces the calls to dynamically preallocate the data in the initialization phase of the application.

Both options push the allocation of memory before the execution of the parallel kernel. Hence, even if the system has insufficient memory at runtime, the error will occur in the initialization phase, so it can be handled in a similar way to a Power-On Self-Test failure[2].

B. Runtime Techniques: Lazy Task Creation

Current OpenMP implementations, e.g., GCC's *libgomp* [9] and LLVM's *kmp* [10], create dynamically the data structures needed to hold OpenMP tasks when the `task` construct is encountered, disregarding if the task is ready for execution or not. We call this *eager task creation*. This methodology allows keeping the creation of the task out of the critical path of the application. However, the amount of memory needed at runtime results bounded to the amount of tasks the application may execute, which may be huge.

We present a new *lazy task creation* mechanism for OpenMP that leverages the information computed at compile-time in order to create tasks only when they become ready, i.e., when all their dependencies are honored. Actually, the maximum number of OpenMP tasks that can become simultaneously ready at runtime is determined by the maximum level of parallelism exposed in the TDG, already computed by our compiler. As a result, our OpenMP framework may guarantee that any ready task will have an available task structure previously allocated by our new compiler features.

However, the compiler may undervalue the amount of parallelism due to the use of a heuristic [24], and hence allocate less task structures than maximum concurrent ready tasks. To address this issue, our runtime implements (a) a preallocated linked list that holds ready tasks with no task structure assigned (named textitgomp_waiting_ready_tasks), and (b) a data-structure per task that holds its data-environment (located in the static TDG). At run-time, when a task is encountered, its data environment is stored. Then, a task structure is only assigned in case all the dependencies are honored, otherwise the task is not created, not wasting memory. If there are not available task structures, the task in inserted in the *gomp_waiting_ready_tasks* list. Then, whenever a task finishes, it first checks this list: if it contains tasks, then one is promoted to the newly freed task structure; if not, tasks that depended on the just finished task are checked and, if their dependencies are honored, a new task structure is assigned to them or they are inserted in the *gomp_waiting_ready_tasks*, depending on the availability. §IV evaluates the performance impact of our OpenMP framework when not enough task structures are available.

Although our approach introduces the creation of the task in the critical path of the application, the overhead of creating a new task is reduced as well, since rather than dynamically allocating a new structure, the runtime simply assigns a structure statically allocated by the compiler. Furthermore, our runtime implementation is not tied to the physical resources, enabling better portability.

C. Putting all Together: Preallocation and Memory Bounding

The amount of structures preallocated to handle ready tasks at runtime is critical for (1) ensuring the best performance of the runtime, and (2) bounding tightly the amount of memory required to exploit the parallel execution. If too few structures are allocated, then ready tasks may be blocked at runtime waiting for an available data structure. On the contrary, if too many data structures are allocated, the runtime may consume unnecessary memory resources.

Our compiler offers different preallocation configurations via two flags:

- `--prealloc`, to use the analyses and lowering needed to preallocate task structures, including the data environment of each task, as described in §III-A1 and §III-A3.
- `--variable="tdg_width"`. This flag, only useful if the former is used (otherwise it is ignored), indicates the computation of the maximum parallelism exposed in the application, as described in §III-A2.

If only `prealloc` is used, then the compiler allocates as many structures as nodes in the TDG, each with the corresponding size for the structure holding the variables needed by the task. Otherwise, *tdg_width* indicates the amount of structures to allocate, and corresponds to the maximum amount of parallelism exposed in the TDG. In this case, the size of the structure for the data environment of each task is the maximum size of the data structures of any task, because any task may potentially use that data structure at runtime.

Our runtime recognizes two new environment variables:

- `OMP_PREALLOC_TASK`, to avoid allocating memory when a task is encountered. Instead, the runtime registers the existence of a new task initializing the values stati-

[2]The *Power-On Self-Test* is a set of procedures that a computer runs each time it is turned on. An error found in the POST is usually fatal, causing the program to stop running, and halts the boot process.

cally allocated in the TDG (data environment, unresolved tasks from which depends, etc.) as described in §III-B.
- `OMP_LAZY_TASK_CREATION`, to create tasks only when they are ready. If the preallocation mechanism is used, then ready tasks are stored in the preallocated structure for such purpose; otherwise, the ready task is dynamically allocated, as in the original version of the runtime.

IV. Evaluation

This section provides a complete evaluation of our OpenMP framework in terms of performance and memory usage with different architectures, runtime configurations and use cases.

A. Experimental Setup

1) Processor Architectures.: We use three architectures from the HPC and the embedded domains: an Intel Xeon Platinum 8160 [25] from the Marenostrum IV supercomputer [26], a TI Keystone II [2], and a GR740 from Cobham Gaisler [27]. The former features 24 cores, the second a quad-core ARM Cortex-A15 host and a 8-core DSP fabric acceleration device, and the latter a quad-core LEON4 SPARC V8. In the TI Keystone II, we only consider the execution on the DSP fabric, and the OpenMP accelerator model is used to offload computation from ARM cores to DSPs.

2) OpenMP Framework.: The three architectures include an OpenMP framework in their SDKs. For the Intel Xeon and the GR740, we use the GNU-GCC version 7.3, implementing the OpenMP 4.5 specification [28]. For the TI Keystone II, we use the framework included in the SDK of the DSPs, corresponding to the OpenMP specification 3.0 [29]. This specification already includes the OpenMP tasking model but with no data dependence support, so we have extended this SDK to support the static generation of the TDG [7]. Furthermore, we use the Mercurium compiler [23], in which we have developed the techniques needed to analyse OpenMP programs and statically generate the data structures to manage the tasking model. Moreover, the runtimes of GCC and the SDK of the DSPs have been modified to support: (1) the data structures statically allocated by Mercurium, and (2) the lazy task creation policy.

3) Application.: We consider four applications from the real-time embedded domain with high-performance requirements: (1) a pedestrian detector based on the computation of a histogram of oriented gradients (HoG), available in the open-source VLFeat library,(2) a Space Time Adaptive Processing (STAP) application used in airborne radars to remove the ambiguity between intrinsic speeds and azimuths of targets, developed by Thales, (3) an image sampling application for infra-red H2RG detector (ESA), and (4) a 3D path planning (r3DPP), used for airborne collision avoidance. The four applications have been parallelized using OpenMP tasks.

Table I characterizes the TDGs of the applications generated with Mercurium, showing the maximum number of nodes (tasks) that form each TDG and the maximum degree of parallelism exposed by each TDG. It is important to remark that the TDG is independent of the execution platform.

TABLE I: Characterization of the TDGs of each application.

	Total Nodes	Max Parallel nodes
HoG Application	3601	45
STAP Application	1123	160
ESA Application	1359	90
3DPP Application	2240	80

B. Performance Speedup

Figure 1 shows the performance speedup on the three considered architectures and two applications, STAP (1a) and ESA (1b) considering two runtimes: (1) the native GCC libgomp included in each processor architecture presented in §IV-A2 (labeled as *OMP*), and (2) our OpenMP framework (labeled as *Qualifiable*) configured to (1) statically allocate as many task data structures as the maximum level of parallelism (see Table I), and (2) use the lazy task creation policy. Speedup is shown varying the number of cores up to the maximum offered in each machine. The number of cores used during the execution is configured via the `num_threads` clause, and the *bind-var* ICV and the `proc_bind` clause have been used to fix threads to cores avoiding other levels of scheduling. The computation considers the arithmetic average time of 50 executions with the exact same data input. Due to space constraints only two of the four applications evaluated are shown in the figure, although all of them have been tested and have a similar behaviour.

The results show that the two OpenMP frameworks present a very similar trend when varying the number of cores in all three architectures, and for all applications (notice the scale difference in the y axis of the charts). Only the STAP shows a slightly better performance in the Intel Xeon processor when the number of threads is higher than 4.

Regarding the scalability of the applications, STAP is limited due to the coarse-grain synchronization directives needed between the execution of consecutive loops, while ESA suffer from dependencies towards a concrete function that is completely serialised during the execution of the application, which reduces considerably the parallelism.

As shown, each application has a different behaviour in terms of parallelism, and each processor has a different architecture. Overall, we conclude that our OpenMP framework is capable of statically allocating all the structures needed to efficiently manage the parallel computation, while providing the exact same performance compared to the OpenMP frameworks provided by Intel Xeon, TI Keystone II and GR740, in which all tasks are dynamically allocated. Moreover, our framework is able to increase the performance in case of the STAP application when executing in the Intel Xeon processor.

It is important to remark that, to preserve backwards compatibility, our modified runtime still contains the original fields and data structures of *libgomp*. This suboptimal data layout may lead to a loss of performance in the configurations that use the static TDG. Nonetheless, our tool still gets at least the same performance as the original tool.

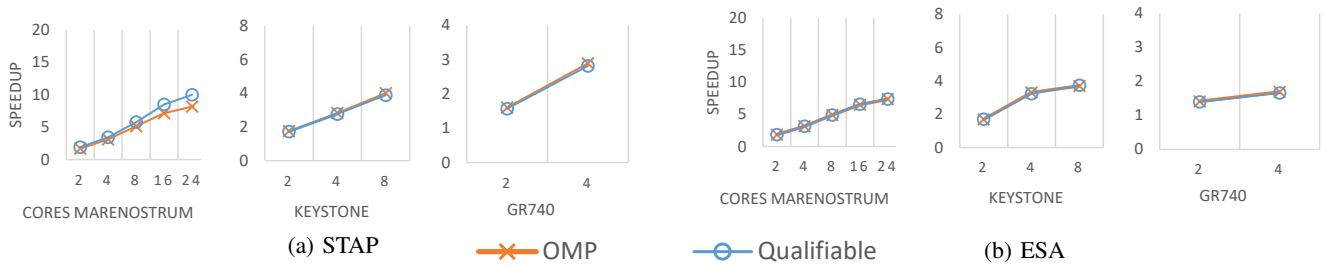


Fig. 1: Benchmarks speedup based on the architecture and the number of cores.

C. Impact of the Number of Task Data Structures

Figure 2 shows the performance speedup of the applications in the Intel Xeon, when reducing the number of preallocated task data structures. As expected, the performance speedup remains the same when the number of task data structures preallocated by the compiler is above the number of computing cores available in the architecture, i.e., 24. From this point on, the performance gets worse due to the lack of available task structures to execute in parallel (the speedup of 3DPP is always 1 because of its bad performance when using 24 cores).

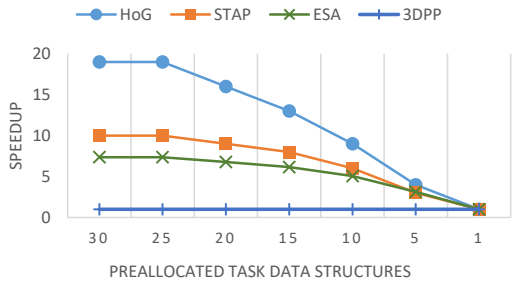


Fig. 2: Preallocated structures impact on speedup.

Overall, the minimum number of task data structures that the compiler has to preallocate to guarantee an efficient parallel execution is the minimum between the number of computing cores available in the underlying architecture and the maximum parallelism exposed in the TDG. However, preallocating a number of structures higher that the number of available cores has no impact on performance (up to a limit indeed), and makes the application binary independent of the specific processor, and so future updates will not require a recompilation of the application.

D. Memory Usage

To evaluate the memory usage of our runtime we focus on two aspects: (1) the amount of dynamic memory consumed by the runtime during the parallel execution, and (2) the total amount of memory consumed by the runtime to manage tasks.

First we analyze in Figure 3 the consumption of dynamic memory of STAP. Each subfigure contains two execution traces (extracted with Extrae [30] and interpreted with Paraver [31]), where the *x* axis represents time and the *y* axis a thread. The trace on top, *Dynamic memory call*, represents calls to *malloc* and *calloc* and different colors represent different allocation sizes; and the trace below, *Parallel*, represents a portion of the parallel execution, particularly its beginning. Subfigure 3a shows the execution using the original implementation of libgomp, where the allocation of memory overlaps with the parallel execution, and subfigure 3b shows the use of preallocated task structures, where the allocation is pushed to the initialization phase, and then starts the parallel execution.

To this day, our implementation still allocates the memory in the heap by using a runtime call. However, with the information computed at compile-time, the compiler could instead create static structures. In that case no dynamic memory would be used at all by the runtime to manage OpenMP tasks.

Second we analyze in Figure 4 the maximum memory space (in KBs) that the different runtime configurations require to allocate the task structures. The graphics show the use of static memory (*STATIC memory*), which is valid for any configuration because it only respects to the static allocation of the TDG, and the use of dynamic memory of three configurations: (1) the compiler preallocates all tasks (*Prealloc full TDG*), (2) it preallocates the TDG's maximum parallelism (*Prealloc max parallelism*), and (3) it preallocates a number of tasks equals to the number of cores (*Prealloc num cores*) (here, the number of available task structures is sufficient to execute in parallel all ready tasks as shown in §IV-C). Notice that the original libgomp implementation uses an amount of memory equivalent to our framework when the full TDG is preallocated.

As the figures reveal, the version that preallocates all tasks, although bounds the memory usage to the number of tasks, results in a very intensive use of memory. Instead, the version that preallocates the maximum amount of parallelism in the TDG or the number of available cores results in a huge fall in the consumption of dynamic memory. Finally, the use of static memory increases with the number of tasks, as the structures to manage the TDG are statically stored by the compiler.

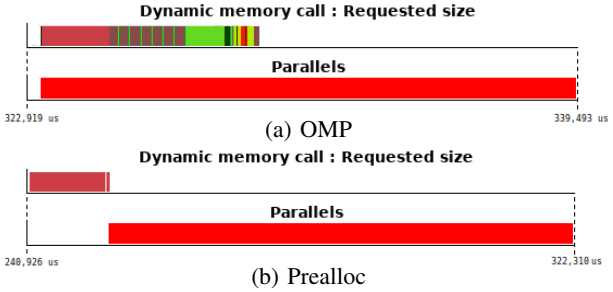
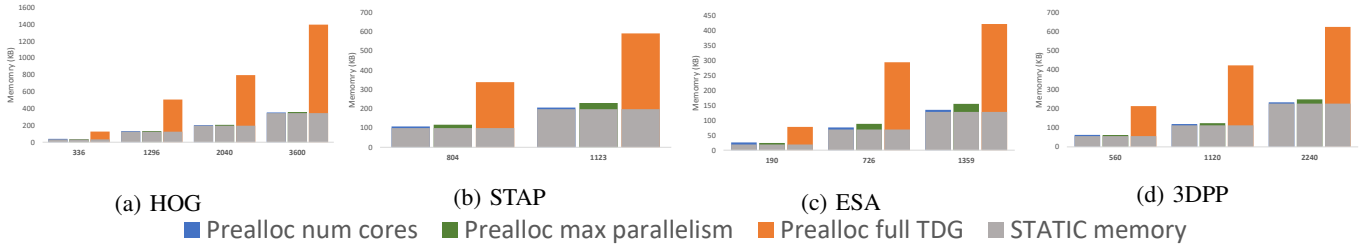


Fig. 3: Dynamic memory usage timeline of the Thales application for different runtimes.



(a) HOG (b) STAP (c) ESA (d) 3DPP

Fig. 4: Memory usage for each application and mechanism, varying the number of tasks (x axis).

V. Conclusion

Despite the proven benefits of OpenMP to develop critical real-time applications, current OpenMP implementations are not suitable due to the intensive use of dynamic memory needed to manage the parallel execution. In that regard, the jeopardy inherent in the use of dynamic memory complicates the qualification of the tool for critical real-time systems. Furthermore, current OpenMP runtimes implement task creation policies that require high amounts of memory, pushing a lot of pressure in the memory requirements, a limited resource in many embedded systems.

This paper proposes a novel OpenMP framework that statically allocates all the structures needed to execute the OpenMP tasking model at compile-time, and implements a *lazy task creation* policy that reduces the amount of on-the-fly task structures required at run-time. This features allow: (1) eliminating the use of dynamic memory in order to manage OpenMP tasks during the execution of the program, (2) bounding the amount of memory required at runtime proportionally to the amount of parallelism exposed in the application.

We have evaluated our framework on three processor architectures from the HPC and the embedded computing domains, executing four real-time applications with high-performance requirements. Results show that our OpenMP framework provides the same or better performance speedup compared with the native framework provided by the processor architectures, while significantly reduces the maximum possible memory consumed to manage the OpenMP tasking model.

Overall, this paper represents a step towards the qualification of an OpenMP framework for critical real-time domains.

References

[1] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *DATE*, 2014.

[2] Texas Instruments, *66AK2Hxx Multicore Keystone II System-on-Chip (SoC)*, 2012. [Online]. Available: www.ti.com/product/66AK2H12

[3] Nvidia, "Jetson agx xavier developer kit," 2019. [Online]. Available: https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit

[4] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, "A real-time scheduling service for parallel tasks," in *RTAS*, 2013, pp. 261–272.

[5] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *CASES*, 2015, pp. 157–166.

[6] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Scheduling and analysis of realtime openmp task systems with tied tasks," in *RTSS*, 2017.

[7] R. E. Vargas, S. Royuela, M. A. Serrano, X. Martorell, and E. Quiñones, "A lightweight openmp4 run-time for embedded systems," in *ASP-DAC*, 2016, pp. 43–49.

[8] OpenMP ARB, "OpenMP 5.0 Specification," 2018. [Online]. Available: www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[9] GNU, "libgomp," 2018. [Online]. Available: gcc.gnu.org/onlinedocs/libgomp/

[10] LLVM, "Openmp* runtime library," 2015. [Online]. Available: https://openmp.llvm.org/

[11] Y. Lin, C. Terboven, D. an Mey and N. Copty, "Automatic Scoping of Variables in Parallel Regions of an OpenMP Program," in *WOMPAT*, 2004, pp. 83–97.

[12] S. Royuela, A. Duran, C. Liao, and D. J. Quinlan, "Auto-scoping for OpenMP Tasks," in *IWOMP*, 2012, pp. 29–43.

[13] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly Detection of Data Races in OpenMP Programs," in *PADTAD*, 2012, pp. 1–10.

[14] S. Royuela, R. Ferrer, D. Caballero, and X. Martorell, "Compiler analysis for openmp tasks correctness," in *International Conference on Computing Frontiers*. ACM, 2015, p. 7.

[15] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B. R. de Supinski, and A. Churbanov, "Towards an error model for openmp," in *International Workshop on OpenMP*. Springer, 2010, pp. 70–82.

[16] S. Royuela, A. Duran, M. A. Serrano, E. Quiñones, and X. Martorell, "A functional safety openmp* for critical real-time embedded systems," in *IWOMP*, 2017, pp. 231–245.

[17] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 2, pp. 149–206, 2001.

[18] I. Puaut, "Real-time performance of dynamic memory allocation algorithms," in *ECRTS*, 2002, pp. 41–49.

[19] S. Jacklin, "Certification of safety-critical software under do-178c and do-278a," in *Infotech@ Aerospace 2012*, 2012, p. 2473.

[20] M. I. S. R. Association *et al.*, *MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013.

[21] B. S. C. (BSC) and E. S. A. (ESA), "Hp4s (high performance parallel payload processing for space)," 2018.

[22] R. Project, "Openmp," 2019, devel.rtems.org/wiki/OpenMP and devel.rtems.org/ticket/2274.

[23] B. S. Center, "Mercurium," 2019. [Online]. Available: pm.bsc.es/mcxx

[24] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, and J. Obdržálek, "The dag-width of directed graphs," *Journal of Combinatorial Theory, Series B*, vol. 102, no. 4, pp. 900–923, 2012.

[25] Intel, "Intel Xeon Platinum 8160," 2018. [Online]. Available: ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2-10-GHz-

[26] BSC-CNS, "Marenostrum iv – technical information," 2019. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[27] C. Gaisler, "Gr740 quad-core leon4 sparc v8 processor," 2019. [Online]. Available: https://www.gaisler.com/index.php/products/components/gr740

[28] OpenMP ARB, "OpenMP 4.5 Specification," 2015. [Online]. Available: www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[29] ——, "OpenMP 3.0 Specification," 2008. [Online]. Available: www.openmp.org/wp-content/uploads/spec30.pdf

[30] BSC, "Extrae," 2019. [Online]. Available: https://tools.bsc.es/extrae

[31] ——, "Paraver," 2019. [Online]. Available: https://tools.bsc.es/paraver